Once upon an API

Michael Kerrisk, man7.org © 2022

mtk@man7.org

3 June 2022, Paris, France
Who?

- Linux *man-pages* project
    - Approx. 1060 pages documenting syscalls and C library
  - Contributor since 2000
  - Maintainer since 2004
  - Comaintainer since 2020

- I wrote a book
- Trainer/writer/engineer
  - [http://man7.org/training/](http://man7.org/training/)
- mtk@man7.org, @mkerrisk

© 2022, Michael Kerrisk

Once upon an API
Who here has a patch in the kernel?
You understand collective responsibility, right?
Why?
Maybe because...

I’d like to see APIs done better

because...

a misdesigned API is (generally) unfixable

(we might break some binary that depends on the broken-ness)

and therefore...

user-space developers must live with broken-ness for a long time
Or, maybe, just tell a story
But I didn’t make up the story. You did.

(Image credit: Piotr Siedlecki)
Story of first feature added in a then-new system call [*]

- `prctl()` (added in 1997 ($t == 0))
- Seems simple
- But, I went down a long rabbit hole one day...

When it comes to APIs, even something that seems simple can turn out to be complicated!

A few thoughts on reducing frequency of design failures in future APIs

[*] I tested various behaviors of this API using this program: http://man7.org/code/proexec/pdeath_signal.c.html
Outline

1. Our story begins 11
2. Missing pieces 17
3. Interactions across the interface 21
4. Surprises: `execve()` 24
5. Surprises: threads 32
6. Missing details: signals 37
7. Surprises: process termination (and subreapers) 39
8. Surprises: threads (again) 43
9. What happened? 49
10. Who owns the interface? 54
11. Insufficient documentation 60
12. Decentralized design often fails us 67
13. In my ideal world... 73
<table>
<thead>
<tr>
<th>Outline</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Our story begins</td>
<td>11</td>
</tr>
<tr>
<td>2 Missing pieces</td>
<td>17</td>
</tr>
<tr>
<td>3 Interactions across the interface</td>
<td>21</td>
</tr>
<tr>
<td>4 Surprises: <code>execve()</code></td>
<td>24</td>
</tr>
<tr>
<td>5 Surprises: threads</td>
<td>32</td>
</tr>
<tr>
<td>6 Missing details: signals</td>
<td>37</td>
</tr>
<tr>
<td>7 Surprises: process termination (and subreapers)</td>
<td>39</td>
</tr>
<tr>
<td>8 Surprises: threads (again)</td>
<td>43</td>
</tr>
<tr>
<td>9 What happened?</td>
<td>49</td>
</tr>
<tr>
<td>10 Who owns the interface?</td>
<td>54</td>
</tr>
<tr>
<td>11 Insufficient documentation</td>
<td>60</td>
</tr>
<tr>
<td>12 Decentralized design often fails us</td>
<td>67</td>
</tr>
<tr>
<td>13 In my ideal world...</td>
<td>73</td>
</tr>
</tbody>
</table>
Since nearly the beginning of \textit{time()}, there has been \texttt{SIGCHLD}

(Signal sent to parent process when child terminates)

(UNIX 4\textsuperscript{th} Edition, 1973)
One day, someone decided the converse might be useful
Hi, Linus. I’ve appended a patch (relative to 2.1.51) which defines a new syscall with the interface:

```c
extern int prctl (int option, ...);
```

Currently the only option which is supported is `PR_SET_PDEATHSIG` [...]

Any child process which does:

```c
prctl (PR_SET_PDEATHSIG, sig);
```

will have `<sig>` delivered to it when its parent process dies. [...] Eventually I hope to see all kinds of `PR_SET_*` and `PR_GET_*` options :-)
I don’t know if Richard Gooch contacted the man-pages maintainer of the time

But Andries Brouwer added documentation in early 1998:

```
PR_SET_PDEATHSIG
sets the parent process death signal of the current process
to arg2 (either a signal value in the range 1..maxsig, or 0 to
clear). This is the signal that the current process will get
when its parent dies. This value is cleared upon a fork().
```
What could go wrong?
Missing pieces

- What about discoverability?
- `prctl(PR_GET_PDEATHSIG, &sig)`
  - Return current setting in `sig`
  - Linux 2.3.15 (Aug 1999, `t+2`)
That’s even simpler...

(but...)
A long time ago I added `PR_GET_PDEATHSIG` to `prctl(2)` to match the existing `PR_SET_PDEATHSIG`. Now that I noticed [the subsequently added `PR_GET_DUMPABLE`] the whole thing looks inconsistent so I suggest to change `PR_GET_PDEATHSIG` so that it is the *return* value of `prctl(PR_GET_PDEATHSIG)` instead of [returning the setting in] the second argument [...]

- `PR_GET_DUMPABLE` returns value as **function result**;
  `PR_GET_PDEATHSIG` returns value via **2nd argument**
  
  ("dumpable" was second `prctl()` operation implemented)

- But at least **we are consistently inconsistent**...

  - Of `prctl()` "get" operations in Linux 5.18 that return `int`: 15 use function result, and 7 use `*arg2`
Outline

1. Our story begins 11
2. Missing pieces 17
3. Interactions across the interface 21
4. Surprises: `execve()` 24
5. Surprises: threads 32
6. Missing details: signals 37
7. Surprises: process termination (and subreapers) 39
8. Surprises: threads (again) 43
9. What happened? 49
10. Who owns the interface? 54
11. Insufficient documentation 60
12. Decentralized design often fails us 67
13. In my ideal world... 73
How does some new feature interact with other parts of the Linux API?
Interactions across the interface

- Surprises may turn up in many places...
- But these areas are often especially rich with surprises:
  - `fork()`
  - `execve()`
  - Signal delivery semantics
  - Threads
  - `exit()` / process termination
- If file descriptors are in play: FDs vs open file descriptions
  - Multiple FDs may refer to same OFD (`dup()`, `fork()`, etc)
  - ⇒ has led to ugly corner cases (e.g., in `epoll(7)`)

© 2022, Michael Kerrisk @mkerrisk

man7.org
Outline

1. Our story begins 11
2. Missing pieces 17
3. Interactions across the interface 21
4. Surprises: `execve()` 24
5. Surprises: threads 32
6. Missing details: signals 37
7. Surprises: process termination (and subreapers) 39
8. Surprises: threads (again) 43
9. What happened? 49
10. Who owns the interface? 54
11. Insufficient documentation 60
12. Decentralized design often fails us 67
13. In my ideal world... 73
Back to the (1998) manual page:

PR_SET_PDEATHSIG
sets the parent process death signal of the current process
to arg2 (either a signal value in the range 1..maxsig, or 0 to
clear). This is the signal that the current process will get
when its parent dies. This value is cleared upon a fork().

When I saw that, I wondered: what about `execve()`?

I didn’t notice that omission until 2014 \(t+17\), but now the
manual page tells us:

This value is preserved across execve(2).
This value is preserved across execve(2).

Maybe, if that detail had been explicitly noted at the start, someone might have noticed a security vulnerability earlier...
Signal permissions

- From *kill(2)*:

> For a process to have permission to send a signal, it must either [have the CAP_KILL capability] or the *real or effective UID of the sending process must equal the real or saved set-user-ID of the target process.*

- Sending signals requires privilege or credential (UID) match
- Can we use `PR_SET_PDEATHSIG` to send a signal to a process we could not otherwise signal?
  - That could be interesting for an attacker...
Scenario 0

fork()--------------------------+
  \ [parent (UID 1000)] [child (UID 1000)]
                  |
                  prctl(PR_SET_PDEATHSIG, SIGHUP)

exit() ---- pdeath sig -----> [child gets signal]

(This is what PR_SET_PDEATHSIG does)
Scenario 1

fork()--------------------------+
| \                          |
| [parent (UID 1000)]        [child (UID 1000)]
|                           |
| prctl(PR_SET_PDEATHSIG, SIGHUP) |
| execve("setuid-root-binary") |
| change all UIDs to 1001     |
| [child is now unsignalable by parent] |

exit() child does not get signal!

- Child execs set-UID-*root* binary that subsequently changes child’s UIDs such that parent can’t signal it
- Consequently, parent-death signal **is not sent** to child
  - (Expected and correct behavior)
Scenario 2 (a security bug)

- Parent execs set-UID binary that gives it same UID as child
- When binary terminates, parent-death signal **is sent** to child
  - I.e., by execing the binary, parent can send signal to child it couldn’t otherwise have signaled
The fix

- In 2007 \((t+10)\), this got fixed
- And documented in 2012 \((t+15)\):

  The parent-death signal setting is cleared for the child of a fork(2). It is also cleared when executing a set-user-ID or set-group-ID binary, or a binary that has associated capabilities; otherwise, this value is preserved across execve(2).

- Clear pdeath signal when execing privileged program
Back to the original patch message:

Any child process which does:
prctl (PR_SET_PDEATHSIG, sig);

will have <sig> delivered to it when its parent process dies.

If “process termination” means “termination of last thread”, this turns out not to be true
- At least not after we got NPTL threading implementation in 2003 ($t+6$)
A bug report

https://bugzilla.kernel.org/show_bug.cgi?id=43300, David Wilcox:

I have a process that is forking to [create] a child process. The child process should not exist if the parent process [exits]. So, I call prctl(PR_SET_PDEATHSIG, SIGKILL) in the child process to kill it if the parent dies. What ends up happening is the parent thread calls pthread_exit, and that thread ends up being the catalyst that kills the child process.

Signal is sent upon termination of creating thread

- I.e., the thread that actually called fork()
- (Rather than when last thread in parent terminates)
That bug report was in 2012! 
\((t+15)\)

(Sometimes, API misdesigns are reported only much later)
A bug that we can’t fix

- And we can’t fix this; Oleg Nesterov in the same bug:

And yes, the current behaviour looks just ugly. The problem is, unlikely we can change it now, this can obviously break the applications which rely on the fact that pdeath_signal is per-thread.

- (I.e., some apps might depend on this strange behavior)

- But at least we can document it (2015, \(t+18\)):

> Warning: the "parent" in this case is considered to be the thread that created this process. In other words, the signal will be sent when that thread terminates [...], rather than after all of the threads in the parent process terminate.

- Actually, the story is even more complicated... (later)
Outline

1. Our story begins
2. Missing pieces
3. Interactions across the interface
4. Surprises: `execve()`
5. Surprises: threads
6. Missing details: signals
7. Surprises: process termination (and subreapers)
8. Surprises: threads (again)
9. What happened?
10. Who owns the interface?
11. Insufficient documentation
12. Decentralized design often fails us
13. In my ideal world...
Does the child get more than a signal?

- So, the child gets a signal; is that all?
- Eventually (2018, \(t+21\)), the manual page noted:

  If the child installs a handler using the `sigaction(2)` SA_SIGINFO flag, the `si_pid` field of the `siginfo_t` argument of the handler **contains the PID of the parent process**

- Note: it’s the **PID** (TGID) of the parent process
  - **Not the TID of the terminating thread!**
  - (You wanted consistency in the API misdesigns?)
Outline

1 Our story begins 11
2 Missing pieces 17
3 Interactions across the interface 21
4 Surprises: `execve()` 24
5 Surprises: threads 32
6 Missing details: signals 37
7 Surprises: process termination (and subreapers) 39
8 Surprises: threads (again) 43
9 What happened? 49
10 Who owns the interface? 54
11 Insufficient documentation 60
12 Decentralized design often fails us 67
13 In my ideal world... 73
What becomes of an orphan?

• Suppose the child continues executing after receiving the parent-death signal...

• Once upon a time, an orphaned child would get adopted by \textit{init} (PID 1)
  • But now, things are different...
Subreapers

- `prctl(PR_SET_CHILD_SUBREAPER, 1)` marks a process as a “subreaper” for any orphaned descendants

A subreaper fulfills the role of `init(1)` for its descendant processes. When a process becomes orphaned, then that process will be reparented to the nearest still living ancestor subreaper.

- If any of my descendants become orphaned, reparent them to me (not to `init`)
- Linux 3.4, 2012 (t+15)
- How do subreapers interact with `PR_SET_PDEATHSIG`?
Subreapers

- Thanks to subreaper mechanism, a child process can have a series of parents
  - (When a subreaper terminates, it’s children are adopted by next ancestor subreaper)
- So, child may get a series of parent-death signals!
- Documented in 2018 \((t+21)\)

The parent-death signal is sent upon subsequent termination of the parent thread and also upon termination of each subreaper process to which the caller is subsequently reparented.
Outline

1. Our story begins
2. Missing pieces
3. Interactions across the interface
4. Surprises: `execve()`
5. Surprises: threads
6. Missing details: signals
7. Surprises: process termination (and subreapers)
8. Surprises: threads (again)
9. What happened?
10. Who owns the interface?
11. Insufficient documentation
12. Decentralized design often fails us
13. In my ideal world...
• Suppose one of those subreaper processes is multithreaded... when does child get the parent-death signal?

• We already entered strange territory a while back...

• So, let’s consider some wild possibilities:
  • (a) When first thread in subreaper terminates
  • (b) When last thread in subreaper terminates
  • (c) When thread group leader in subreaper terminates
  • (d) Upon termination of each thread in subreaper
But, like those school tests...
(e) none of the above
My understanding of kernel’s `find_new_reaper()`:

- Child processes are parented by individual threads
- When a thread terminates, its children are reparented
  - And child gets pdeath signal, if it requested it
  - If parent was single threaded ⇒ reparent to next ancestor subreaper
- Else if multithreaded, reparent to another thread in parent!
  - **Search for new parent is in order of thread creation**, starting with thread group leader
  - Reparenting to another thread **might happen multiple times** as threads terminate in parent
- Same behavior for original parent of process that used `PR_SET_PDEATHSIG`
Do I dare to document this?

(So far, I did not)

(But, given enough time, users will invent every possible use case for an API, or write programs that accidentally depend on obscure details of API behavior)

(Even if we don’t document it...)
What the original author wanted:

Child should get a signal when parent terminates
What we actually got...

- If parent is multithreaded, child gets signal when creating thread terminates
- Child may get multiple signals if parent is multithreaded
  - # of signals depends on order of thread creation in parent!
  - Each signal has same `si_pid` value
- Accidental exposure of details of kernel’s implementation of process management
- Child gets multiple signals if there are ancestor subreapers
  - And if those subreapers are multithreaded, see above...
- A security bug (signal a process owned by another UID)
  - Now fixed
- The start of an API inconsistency (`prctl()` “get” operations)
Clearly, many of these behaviors were unintended.
What went wrong?

- No one person/group owns the interface
- No/insufficient documentation
- Insufficient consideration of interaction with other parts of interface
- Behavior evolved with the addition of other interfaces / kernel features
  - Some of these behaviors almost certainly changed over time
- Decentralized design often fails us
Outline

1. Our story begins
2. Missing pieces
3. Interactions across the interface
4. Surprises: `execve()`
5. Surprises: threads
6. Missing details: signals
7. Surprises: process termination (and subreapers)
8. Surprises: threads (again)
9. What happened?
10. **Who owns the interface?**
11. Insufficient documentation
12. Decentralized design often fails us
13. In my ideal world...
Who owns the interface?

IOW: who gets to say what the interface contract is?

The answer isn’t simple...

(\url{http://man7.org/conf/lpc2008/who_owns_the_interface.pdf})
Do the kernel developers define the interface contract?

- It “must” be the kernel developers, right?
  - They write the code!
- But...
  - What if **implementation deviates** from intention? (A bug)
  - What about **unforeseen uses** of interface?
  - **C library wrappers** mediate between kernel and user space
Do the glibc developers define the interface contract?

- Is it the glibc developers?
- Glibc provides wrappers for most system calls
  - Sometimes wrappers change or add behavior
- But...
  - In many cases, wrapper is trivial (no behavior change)
  - Sometimes, it's a long time before wrapper lands in glibc
    - 18 years until `gettid()` got a wrapper
      - https://sourceware.org/bugzilla/show_bug.cgi?id=6399
Does documentation define the interface contract?

- **man-pages** documents kernel APIs
  - Goal: document what kernel guarantees to user space
  - Documentation can act as specification, describing developer’s intention
    - Allows testing for difference between implementation and intention

- But...
  - Many things remain undocumented
  - Sometimes implementation is right and docs are wrong :-(

©2022, Michael Kerrisk @mkerrisk
Do user-space developers define the interface contract?

- Is it user-space developers?
- How could it possibly be the users?
- Given enough time, users collectively discover every possible detail of the API
  - *Deliberately*: user discovers API behaviors and explicitly makes use of them
  - *Accidentally*: user writes code that implicitly depends on an API behavior (including API bugs)
- User code may depend on behaviors that implementer hadn’t considered/was unaware of
  - Ancient example: oddball use cases for files with permissions such as `rw----r--`!
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Our story begins</td>
<td>11</td>
</tr>
<tr>
<td>2  Missing pieces</td>
<td>17</td>
</tr>
<tr>
<td>3  Interactions across the interface</td>
<td>21</td>
</tr>
<tr>
<td>4  Surprises: <code>execve()</code></td>
<td>24</td>
</tr>
<tr>
<td>5  Surprises: threads</td>
<td>32</td>
</tr>
<tr>
<td>6  Missing details: signals</td>
<td>37</td>
</tr>
<tr>
<td>7  Surprises: process termination (and subreapers)</td>
<td>39</td>
</tr>
<tr>
<td>8  Surprises: threads (again)</td>
<td>43</td>
</tr>
<tr>
<td>9  What happened?</td>
<td>49</td>
</tr>
<tr>
<td>10 Who owns the interface?</td>
<td>54</td>
</tr>
<tr>
<td>11 Insufficient documentation</td>
<td>60</td>
</tr>
<tr>
<td>12 Decentralized design often fails us</td>
<td>67</td>
</tr>
<tr>
<td>13 In my ideal world...</td>
<td>73</td>
</tr>
</tbody>
</table>
The original PR_SET_PDEATHSIG documentation (1998)

PR_SET_PDEATHSIG sets the parent process death signal of the current process to arg2 (either a signal value in the range 1..maxsig, or 0 to clear). This is the signal that the current process will get when its parent dies. This value is cleared upon a fork().

Compare that with what we have now....
Set the parent-death signal of the calling process to arg2 (either a signal value in the range 1..maxsig, or 0 to clear). This is the signal that the calling process will get when its parent dies.

Warning: the "parent" in this case is considered to be the thread that created this process. In other words, the signal will be sent when that thread terminates (via, for example, pthread_exit(3)), rather than after all of the threads in the parent process terminate.

The parent-death signal is sent upon subsequent termination of the parent thread and also upon termination of each subreaper process (see the description of PR_SET_CHILD_SUBREAPER above) to which the caller is subsequently reparented. If the parent thread and all ancestor subreapers have already terminated by the time of the PR_SET_PDEATHSIG operation, then no parent-death signal is sent to the caller.

The parent-death signal is process-directed (see signal(7)) and, if the child installs a handler using the sigaction(2) SA_SIGINFO flag, the si_pid field of the siginfo_t argument of the handler contains the PID of the parent process.

The parent-death signal setting is cleared for the child of a fork(2). It is also (since Linux 2.4.36 / 2.6.23) cleared when executing a set-user-ID or set-group-ID binary, or a binary that has associated capabilities (see capabilities(7)); otherwise, this value is preserved across execve(2). The parent-death signal setting is also cleared upon changes to any of the following thread credentials: effective UID, effective GID, filesystem UID, or filesystem GID.
Many of these details were documented only long after the fact...

If they had been documented at the time, would we have done things differently?
Documentation

- Documentation can come in various forms...
  - A *man-pages* patch or a
  - A really well written commit message
- If **documentation** was written as API is implemented/modified, it might have helped reviewers spot these problems
  - Documentation lowers bar for code review
    - (And provides a spec for testing)
Documentation is a time multiplier

(Photo credit: Robert Bye)
I’ll just ignore the many problems that insufficient documentation creates for API consumers.
Inconsistencies and surprises

- Inconsistencies: `PR_GET_PDEATHSIG` vs `PR_GET_DUMPABLE`
  - Return info via function result or via an argument?
- 2003 addition of NPTL almost certainly changed the behavior of `PR_SET_PDEATHSIG`
  - Child gets signal when creating thread terminates
  - Child may get multiple signals as threads in parent terminate one by one
- 2012 addition of `PR_SET_CHILD_SUBREAPER` magnified the previous point
- In each case, problem was failure to see the bigger picture
PR_SET_PDEATHSIG is a small lesson in the school of “we don’t do decentralized design well”
We’ve had much harsher lessons

Control groups v1,

overloaded \texttt{CAP\_SYS\_ADMIN} capability,

...
We have too few eyes looking at the big picture

Not enough people with motivation, time, and knowledge to consider things such as API consistency and interactions across the interface
Why isn’t there a paid kernel user-space API maintainer(s)?
In my ideal world, many things would happen

(But, I’ll focus on just a few)
Features should have real users

- No new API would be merged without a real-world app that provides a first test of the design (and implementation)
- Many times, real users started using API only after it was merged into kernel
  - Then we discovered the (usually unfixable) design problems
- Example sad story: *inotify*
  - [https://lwn.net/Articles/605128/](https://lwn.net/Articles/605128/)
Commit messages

- Every commit message, but especially those that change interfaces would
  - Explain **why** the change is being made
  - Include explanations of why features are included
  - Include explanations of why features are not included
  - Include a **version history** that explains how patch evolved over time
    - (That often helps with two preceding points)
  - Include **URLs referring to mailing list discussions**
- It’s not **so** hard, and it will make your patches better...
  - Take a lesson from Christian Brauner
    - E.g., 3eb39f4793 and 7f192e3cd3 are a joy to read
A *man-pages* patch would be written in parallel with development of new API
- Not as an after-thought

Documenting an API:
- Is a great trigger for developer to reconsider their design concept
- Lowers bar for reviewers to understand (and therefore comment) on your patch

And of course, end users will thank you for that documentation
And of course many other things...

- Writing tests
- Engaging with glibc maintainers
- CCing linux-api@vger.kernel.org
Thanks!

Michael Kerrisk, Trainer and Consultant
http://man7.org/training/

mtk@man7.org   @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/

THE LINUX PROGRAMMING INTERFACE
A Linux and UNIX® System Programming Handbook
MICHAEL KERRISK